Scalia: An Adaptive Scheme for Efficient Multi-Cloud Storage

Thanasis G. Papaioannou, Nicolas Bonvin and Karl Aberer
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

***Abstract*—A growing amount of data is produced daily resulting in a growing demand for storage solutions. While cloud storage providers offer a virtually infinite storage capacity, data owners seek geographical and provider diversity in data placement, in order to avoid vendor lock-in and to increase availability and durability. Moreover, depending on the customer data access pattern, a certain cloud provider may be cheaper than another. In this paper[1], we introduce Scalia, a cloud storage brokerage solution that continuously adapts the placement of data based on its access pattern and subject to optimization objectives, such as storage costs. Scalia efficiently considers repositioning of only selected objects that may significantly lower the storage cost. By extensive simulation experiments, we prove the cost-effectiveness of Scalia against static placements and its proximity to the ideal data placement in various scenarios of data access patterns, of available cloud storage solutions and of failures.**

I. Introduction

Cloud providers are offering efficient on-demand storage solutions that can virtually scale indefinitely. Many public cloud storage providers are already available in the market, such as Amazon S3 (http://aws.amazon.com/s3), Google Storage (http://code.google.com/apis/storage), Microsoft Azure (http:// microsoft.com/windowsazure) or RackSpace CloudFiles (http://rackspace.com/cloud) and one may expect new providers to appear in the coming years. The offers in terms of pricing among providers vary significantly and may change over time to adapt to the market. Choosing the best-suited or cheapest provider for your data implies knowing in advance the access pattern to the data. Data that is rarely accessed should be stored at a cloud provider mainly with a low storage price, regardless of its access prices. On the other hand, a very popular data may be hosted on a provider with attractive price for the outgoing bandwidth. In most cases, it is difficult to know in advance the access pattern of a data item, and therefore one needs an adaptive solution to choose the most cost-efficient provider.

However, finding a suitable provider based on the access pattern of the data is not enough. A provider may end its business or suddenly increase its pricing policy. There exist many other technical as well as non-technical (e.g., boycotting a provider) reasons a user may want to change its provider. Therefore, in order to safely host its data and minimize the impact of the migration to a new provider, a user needs to proactively avoid vendor lock-in (i.e., being dependent on a specific service vendor with substantial switching costs) and ensure high durability and availability by geographic diversification of the data placement (e.g., the recent Amazon outage reminds us not to put all eggs in one basket, see http://aws.amazon.com/message/65648).

Abu-Libdeh *et al.* underline in [2] the advantages of splitting a data object (e.g., a file) into chunks and storing them across several storage providers, in order to reduce costs and avoid vendor lock-in. However, a more adaptive approach is required to cope with dynamically changing conditions, such as varying data access patterns, evolving pricing policies, new providers arrival, as well as providers' bankruptcy. Moreover, different data access patterns result in different optimal sets of providers in terms of charging.

In this paper, we introduce *Scalia*, a system that continuously adapts the placement of data among several storage providers subject to optimization objectives, such cost minimization. Our system combines the following unique and novel characteristics:

1) Adaptive data placement based on the real-time data access patterns, so as to minimize the price that the data owner has to pay to the cloud storage providers given a set of customer rules, e.g., availability, durability, etc. Other optimization goals for data placement are also conceivable, such as a) maintaining a certain monthly budget by relaxing some constraints, such as lock-in or availability, or b) minimizing query latency by promoting the most high-performing providers.
2) Compliance with the rules set by customers for data, such as data durability, data availability and level of vendor lock-in.
3) Orchestration of a non-static set of public cloud and corporate-owned private storage resources.
4) A robust distributed architecture for its implementation that is able to handle a large number of objects stored, which are accessed by a large number potential users.

The remainder of this paper is organized as follows: In Section II, we discuss the problems of vendor lock-in and paying unfairly high prices when fixed sets of cloud storage providers are employed. In Section III, we describe our *Scalia* brokerage architecture, the adaptive data placement mechanism, the data

[1]Partially supported by the EU project OpenIoT (ICT 287305).

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/$31.00 © 2012 IEEE

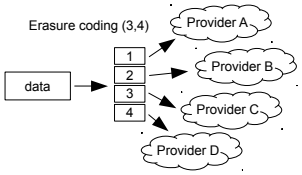

Fig. 1. Erasure coding (m, n): any m-subset of the n chunks contains a complete copy of the data.

caching layer, the metadata storage layer and *Scalia* actions in data read/write operations. In Section IV, we assess the effectiveness of our approach for cost-effective data placement in case of different data access patterns and of cloud resource addition/failure. In Section V, we present some related work, and finally, in Section VI, we conclude our work.

II. Motivation

A. Avoiding Vendor Lock-in

1) Erasure Coding: In order to avoid vendor lock-in, data has to be hosted by multiple storage providers. However, despite being simple and reactive, storing full replicas of the same data is too costly [4], [5]. With the aid of erasure coding (m, n) [3], a data can be split into n chunks ($n > m$), where any m-subset is sufficient to reconstruct a complete copy of the data. The rate $r = \frac{m}{n} < 1$ of an erasure code is the fraction of chunks required to rebuild the original data. The disk space needed to store an r-encoded object increases by a factor of $\frac{1}{r}$. In Figure 1, the original data can be rebuilt with the chunks stored at any 3 of the 4 cloud providers. For example, RAID 1 (mirroring without parity or striping) can be achieved by setting $m = 1$, while RAID 5 (block-level striping with distributed parity) can be described by $(m = k, n = k + 1)$, where $k \geq 3$.

Redundant striping presents several advantages. First, it allows to tolerate up to $n - m$ provider outages, hence greatly improving the durability as well as the availability of the stored data. The user may also choose how to recover from a provider failure. One might decide to reconstruct the missing chunks from the other providers and store them to new providers, or on the other hand, one might decide to ignore the failure and wait for the provider to recover. Second, striping provides a finer granularity than full replication, which permits to read from the cheapest provider or to move a restricted number of chunks to a cheaper provider. Also, it gives a better control on the cost by allowing to store and serve data from public providers as well as private storage facilities.

B. Paying a Fair Price

Given customer (i.e., data owner/producer) requirements (possibly differentiated per data item), such as data durability, data availability or independence from cloud providers to avoid vendors lock-in, it then becomes a non-trivial task to find the cloud storage provider(s) or combinations of cloud storage providers that offer the best price to store users' data. To make things worse, the ratio of read/write operations of a data object over a period of time (i.e., the data access pattern) affects the resulting charging for the customer, as providers implicitly promote certain access patterns with their pricing policies. *Scalia* provides an engine that optimizes the placements of data chunks following the rules set by the data owner, while also taking into account the access patterns of the data in order to compute the cheapest provider set. A default rule, rules per data object classes or rules per data object can be defined in *Scalia* (e.g., using an API or a Web interface), so as to specify the availability, the durability, the geographical zone(s) and the lock-in factor of the data, as described in Table 2. The lock-in factor $obj[lockin] \in (0,\ 1]$ of a data object obj is defined as:

$$obj[lockin] = \frac{1}{N_{obj}}, \tag{1}$$

where N_{obj} is the number of minimum distinct providers where the data object obj will be stored.

Name	*Durability*	*Availability*	*Zones*	*Lock-in*
Rule 1	99.9999	99.99	EU, US	0.3
Rule 2	99.999	99.99	EU	1
Rule 3	99.99	99.99	all	0.2

Fig. 2. Example of storage rules.

Prices in USD per GB for storage, bandwidth in and out, or in USD per 1000 requests for the operations

Description	*Name*	*Durability*	*Avail.*	*Zones*	*Storage*	*Bdw in*	*Bdw out*	*Ops*
Amazon S3 (High)	S3(h)	99.999999999	99.9	EU, US, APAC	0.14	0.1	0.15	0.01
Amazon S3 (Low)	S3(l)	99.99	99.9	EU, US, APAC	0.093	0.1	0.15	0.01
Rackspace CloudFiles	RS	99.9999	99.9	US	0.15	0.08	0.18	0.0
Microsoft Azure	Azu	99.9999	99.9	US	0.15	0.1	0.15	0.01
Google Storage	Ggl	99.9999	99.9	US	0.17	0.1	0.15	0.01

Fig. 3. Example of providers.

Given the users' rules, the engine stores the user data at the cheapest provider set among the complete range of possible alternatives, and continuously adapts the data placement to match the data access pattern. For example, a user looking to store non-critical and ephemeral data will not be interested in avoiding vendor lock-in or storing its data to a high durability provider. On the other hand, if one wants to store critical data over a long period of time, vendor lock-in as well as durability become serious issues. Cold data may be stored at providers offering the cheapest storage price, regardless of the price of bandwidth or of operations, while popular data should be stored to providers showing low prices regarding outgoing bandwidth. By only specifying simple rules, a user should be able to always pay a fair price according to his real needs.

III. Scalia: Multi-Cloud Storage

In this section, we describe *Scalia* in detail and present its complete architecture, which enables to aggregate public cloud storage providers and private storage resources. *Scalia* can run directly at the customer premises as an integrated hardware and software solution (i.e., an appliance) or can be deployed as a hosted service across several datacenters, putting the emphasis on providing a scalable and highly-available architecture with no single point of failure, able to guarantee higher availability than the storage providers. In the first deployment model, the appliance is located directly in the customer's data center, with the advantage of not introducing

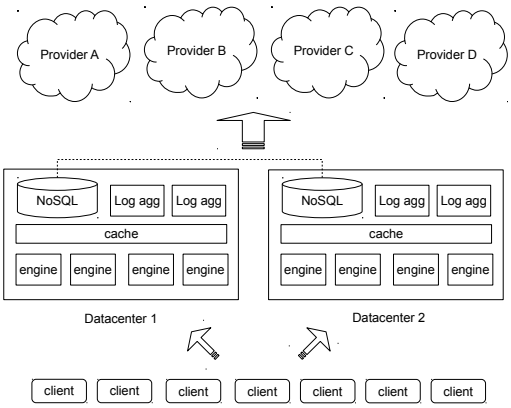

Fig. 4. Multi-datacenter architecture.

additional network latency, not having to pay any extra service fee and not being dependent on the availability of the hosted service. On the other hand, when *Scalia* is accessed as hosted service, a customer does not need to install any additional hardware or software and will pay only service fees. The hosted service can be operated by an independent broker for multiple customers.

In Figure 4, for simplicity, we consider *Scalia* as a hosted service in a setup consisting of only a pair of datacenters. A client can send requests indifferently to each datacenter. The *Scalia* brokerage system consists of three layers: a layer of stateless engines, a caching layer and a database layer. The engines provide an Amazon S3-like interface (i.e. compatible to existing solutions employed by the end-users), where the users can put, get, list and delete their data using a key-value data model. The engines are responsible for computing the best provider set according to the user requirements, for maintaining the cost-effective data placement using the access history of the data, for splitting and storing the chunks at the most suitable providers, for reconstructing the data from the chunks and finally for deleting the data. Each engine works independently and does not keep a state. This allows this layer to scale linearly by just adding new engine components. The caching layer is not mandatory, yet if employed, it greatly improves the performance for read operations of popular data and reduces the corresponding costs for data fetching. The database layer is responsible for hosting the metadata of the data stored in the remote storage providers, and to store their access statistics.

A. Engine Layer

The engine acts as a proxy between the client and the cloud storage providers, offering a unified API to all providers, including data storage to private resources. Mainly, it is responsible for storing the chunks of data to the best providers according to the optimization goals, and serving the data either directly from the cache or by reconstructing it using the chunks stored at the remote providers.

The engine also tries to maintain the optimality of the chunk placement of an object obj, by periodically recomputing the best provider set using the data access statistics of the last $|D_{obj}|$ sampling periods, where $D_{obj} \subset H_{obj}$ is referred to as *decision period* and corresponds to the period of historical access statistics used to compute the chunk placement that is *expected* to be optimal. The access statistics of a data object obj are kept in the history H_{obj}. The sampling period s is a time period where the statistics per object are collected and aggregated, typically 1 hour. Knowing the recent access history of a data permits to precisely adjust the set of providers, as we can reasonably suppose that the access pattern of the data in the near future will be similar to the current. Choosing a large decision period allows to predict the access pattern farer in the future, and thus permits to make better placement choices in the long run. However, imagine that the chunks of a data object were placed based on the assumption that the object would be stored for at least 6 months, and the object was in fact deleted after 1 week. The chosen placement would have been probably wrong, resulting in higher costs for the end user. Thus, the decision period D_{obj} has to be dynamically adjusted as it depends on the lifetime of the object, the burstiness of its access pattern and the resulting economic impact of the latter.

In practice, it is determined based on a dichotomic search between 0 and $min(TTL_{obj}, H_{obj})$, where TTL_{obj} is the time left to live of the data object obj, as described below. When a periodic optimization procedure begins (as will be described in Section III-A3), historical access statistics of length $\frac{D_{obj}}{2}$, D_{obj}, and $2 * D_{obj}$ are considered in parallel (i.e. coupling) when computing the best set of providers using Algorithm 1. D_{obj} is then updated to the decision period based on which the cheapest set of providers is found among the three best sets. This approach for updating D_{obj} is applied every T optimization procedures. Initially, $T = 1$ and whenever D_{obj} is found to be adequate, T is doubled, otherwise, T is reset to the initial value, i.e., $T = 1$. The maximum value of T can be considered to be a period of weeks. We consider here two approaches to determine TTL_{obj}: a) An indication of the object lifetime may be provided by the end user at write time, allowing *Scalia* to make the best choices for chunk placement. b) Otherwise, *Scalia* employs statistics collected from all data objects to find out the most probable lifetime of a certain data item, as explained in the next subsection.

1) Classification of Objects: An object belongs to a class of objects determined by its metadata such as size or MIME type. The class of an object $C(obj)$ is derived using a simple hash of relevant metadata:

$$C(obj) = MD5(obj[mime] \mid discretize(obj[size]))$$

where $discretize()$ is a function which rounds a number to a close integer (e.g., the size of an object is rounded up to the closest megabyte).

For every class of object, *Scalia* collects statistics regarding the resources used (i.e., bandwidth in and out, operations, deletion time, ...) and computes the lifetime distribution of the class, in order to dynamically assign a satisfying value for the decision period D_{obj} and to predict the lifetime of a new object at the time of insertion. As shown in Figure 5, given the deletion time of the objects of a certain class (left),

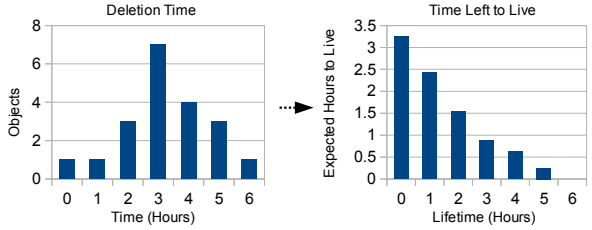

Fig. 5. Time left to live for a class of objects, as computed by the statistics. The class contains 20 objects, whose lifetime varies from 0 to 6 hours.

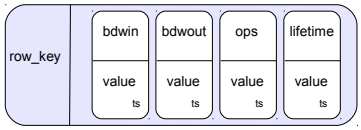

Fig. 6. Statistics are used to improve the first placement of an object.

one can compute the most probable time-left-to-live for an object (right). For example, at insertion time, the lifetime of an object of that class is expected to be 3.25 hours, while a 2 hours old object is expected to live for 1.55 hour more. The lifetime distribution of the classes of objects stabilizes after a training phase, and thus does not incur extra computing costs. The training phase should span the lifetime of some objects belonging to an object class. Initial training can be omitted and replaced by dynamic adjustment only, if initial estimates on the lifetime distribution of the object classes are known to the data owner. The statistics and distributions of the classes of objects are periodically refreshed using map-reduce jobs in the database layer.

2) Placement Algorithm: The time is divided into sampling periods. In current public cloud storage system, this period usually corresponds to 1 hour. For a sampling period s_i at time i, statistics of a data object obj are collected, such as the used storage $s_i[storage]$, the incoming bandwidth $s_i[bwdin]$, the outgoing bandwidth $s_i[bwdout]$ as well as the number of operations $s_i[ops]$. Let $H(obj) = \{s_{t-0}, s_{t-1}, s_{t-2}, \ldots, s_{t-|D_{obj}|}\}$ be the list of access history statistics of the data object at time t.

At insertion time, a data object has obviously no access history, and therefore the provider set chosen by the placement algorithm might change in a near future, when the data object has some accesses. Therefore, *Scalia* uses the statistics collected for the class of the object to determine the statistically best set of providers for this new object. Intuitively, a large archive file is most probably a backup, which will not be read often. On the other hand, a small image (such as a logo) will have plenty of read operations. The optimal set of providers for the aforementioned two examples will be different. Thus, thanks to the statistics collected for each class of objects, the probability that the first placement is already optimal increases. As depicted in Figure 6, given $row\_key = C(obj)$, the placement algorithm has access to the most probable values regarding the resources that the new object obj will use and its lifetime, and therefore is able to make the best possible placement at this early point. Let $P(obj) = \{p_i\}$ be the set of storage providers (both public and private) available for storing the data object obj, with $|P|$ being the total number of providers. A data object has to satisfy several properties contained in the service level agreement (SLA) with the user, such as the minimum durability $obj[durability]$, the minimum availability $obj[availability]$ and the lock-in ratio $obj[lockin]$. Note that the algorithm is not restricted only to these user requirements.

Algorithm 1 Get the best provider set for storing the chunks of a data object obj using its access history $H(obj)$.

$price \leftarrow MAX\_DOUBLE$; $providers \leftarrow \{\}$;
$threshold \leftarrow 0$; $combs \leftarrow \{\}\{\}$;
$combs \leftarrow getAllCombinations(P(obj))$;
for all $pset \in combs$ **do**
  $lockin \leftarrow 1/|pset|$
  continue if $lockin > obj[lockin]$
  $th \leftarrow getThreshold(pset, obj[durability])$
  continue if $th \leq 0$
  $av \leftarrow getAvailability(pset, th)$;
  continue if $av < obj[availability]$
  $pr \leftarrow computePrice(pset, H(obj))$
  if $pr < price$ **then**
    $price \leftarrow pr$
    $providers \leftarrow pset$
    $threshold \leftarrow th$
  end if
end for
return $\{providers, threshold\}$

Algorithm 2 *getThreshold()* function: compute the largest threshold given the set of providers $pset$ and the required durability dr.

Require: $pset, dr$
$dura \leftarrow 0$
$failuresOK \leftarrow -1$
$combs \leftarrow \{\}\{\}$
while $dura < dr$ && $failuresOK < |pset|$ **do**
  $failuresOK \leftarrow failuresOK + 1$
  $upP \leftarrow 0$
  $combs \leftarrow getCombinations(pset, failuresOK)$
  for $comb \in combs$ **do**
    $upPComb \leftarrow 1$
    for all $p \in pset$ **do**
      if $p \in comb$ **then**
        $upPComb \leftarrow upPComb * (1 - p[durability])$
      else
        $upPComb \leftarrow upPComb * p[durability]$
      end if
    end for
    $upP \leftarrow upP + upPComb$
  end for
  $dura \leftarrow dura + upP$
end while
return $|pset| - failuresOK$

Algorithm 1 describes how to compute the best provider set for storing the chunks of a data object obj based on its access history $H(obj)$. The function $getAllCombinations()$ returns the list of every combination of the $|P|$ providers available for an object. Provider constraints in chunk size are taken into account in data placement as follows: two choices are evaluated in terms of expected price, namely inclusion of the constraining cloud provider (smaller chunks) vs. exclusion of the constraining cloud provider (larger chunks). As described in Algorithm 2, the largest value of m, as defined in Subsection II-A1, for a set of providers is given by $getThreshold()$, so as to satisfy the durability constraint of the object. Let us recall that having a value as large as possible for m, referred to as *threshold*, reduces the vendors lock-in and minimizes the storage overhead introduced by the erasure coding of the object. In Algorithm 2, starting from zero, the number of failed providers is increased until the durability constraint $obj[durability]$ (dr in Algorithm 2) is no more satisfied by comparing dr with the probability that the object obj can be reconstructed from the non-failed providers according to the SLA durability of each provider. When the threshold is equal or less than zero, the set of providers is not able to satisfy the durability constraint. The function $getAvailability()$ computes the availability of the object offered by the set of providers passed as parameter according to their SLA, in order to be compared with the minimum availability requirement $obj[availability]$ of the object. The availability value av is obtained by computing the probability of the object to be successfully reassembled when up to th providers are unreachable. Finally, given the access history of an object, the function $computePrice()$ returns the expected cost that a user may have to pay in the next decision period if the object is stored at the provider set taken as parameter.

The complexity of the Algorithm 1 is $O(2^{|P|})$, where $|P|$ is the number of cloud storage providers. However, only the minimally feasible solutions have to be explored, which are much fewer than $2^{|P|}-1$. As there are currently only a few (less than 15) cloud storage providers available on the market, finding the optimal solution based on the access statistics is still computationally feasible. If the number of providers increases, then suboptimal solutions have to be considered. Actually, this optimization problem resembles the multi-dimensional knapsack problem [6], which is NP-complete. In the knapsack problem, one has to maximize the value of items in a knapsack, while respecting a maximum weight constraint. In our case, we want to minimize price, while satisfying the minimum availability, durability, and lock-in constraints. For any fixed number of constraints, the knapsack problem does admit a pseudo-polynomial time algorithm [6] (similar to the one for basic knapsack) and a polynomial-time approximation scheme. Such a heuristic would render Scalia highly scalable. The presentation of this algorithm is omitted for brevity reasons.

3) Periodic Optimization: Recomputing the placement of every data item may become costly as the number of unique data objects can be very large (e.g., Amazon S3 is reported to store more than 339 billion objects as of June 2011). Iterating over all entries (i.e., a full table scan) is obviously not a scalable solution. Note that the provider set of an object will change only if its access history varies significantly or if the set of storage providers $P(obj)$ changes. Therefore, detecting the changes of the access history pattern of the objects and only optimizing the placement of the objects that may have a new economically-efficient provider set greatly reduces the amount of work and resources needed to continuously ensure that every object is optimally placed. It also permits to run the optimization procedure often, so that the system reacts fast. Moreover, the operational and computational complexity of the placement optimizations should be kept as low as possible in order the solution to remain scalable when the number of managed objects increases.

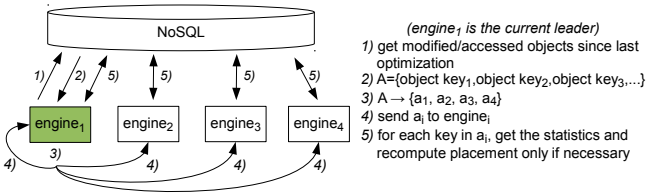

Fig. 7. Periodic Optimization.

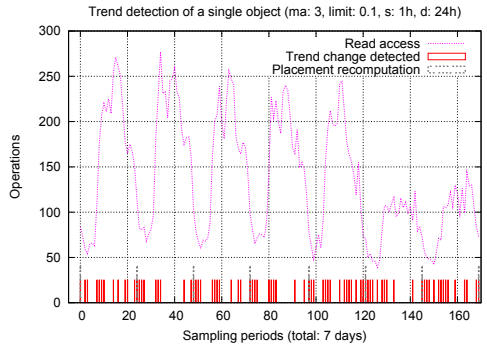
Trend detection of a single object (ma: 3, limit: 0.1, s: 1h, d: 24h)
Read access
Trend change detected
Placement recomputation
Operations
Sampling periods (total: 7 days)

Fig. 8. Trend detection using a threshold limit of 0.1, a sampling period of 1 hour and a decision period of 1 day (24 hours).

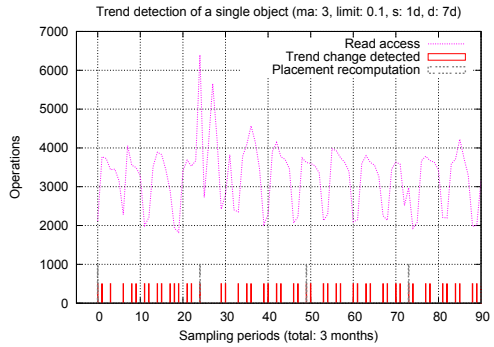
Trend detection of a single object (ma: 3, limit: 0.1, s: 1d, d: 7d)
Read access
Trend change detected
Placement recomputation
Operations
Sampling periods (total: 3 months)

Fig. 9. Trend detection using a threshold limit of 0.1, a sampling period of 1 day and a decision period of 1 week (7 days).

Periodically (e.g., every 5 minutes), *Scalia* starts the optimization procedure. At time t, a new optimization procedure o_t starts: a leader, elected among all engines from all datacenters, retrieves from the statistics database the set $A = \{obj_i\}$ of

object keys that have been accessed or modified after the last optimization procedure o_{t-1}. The leader splits A into $|E|$ subsets of equal size, where $E = \{e_i\}$ is the set of all engines from all datacenters. A subset a_i of keys is assigned to each engine $e_i \in E$. For every object key in a_i, an engine e_i will determine whether the access history pattern of the object has changed or not, by using a $detect()$ function described as follows: In order to detect a changing access pattern at time t, a statistics window of size $w = 3$ sampling periods is employed. High values of w detect trend changes in long time scales, while small values of w should be employed for detecting frequent trend changes. The algorithm also takes as input a threshold $limit$ (e.g. 10% was experimentally found to perform adequately), which is dynamically determined as the minimum momentum (i.e. change in the simple moving average) per object class that would result into a different best set of providers. Momentum is employed for trend change detection, but alternative approaches (e.g. regression models, neural networks, etc.) and other indicators (e.g. rate of change, stochastic indicator, etc.) are also possible.

Only if the access history pattern has changed considerably (based on $limit$), the engine will recompute the placement of the object using Algorithm 1. If a better provider set is found and if the cost of migration is *covered* by the benefits of migrating to the new provider, it will migrate the chunks accordingly. The placement of objects with no access or a non-varying access pattern will not be recomputed. Figure 8 and 9 show when the object placement is recomputed, given a real website access pattern (the website has around 2500 visitors per day mainly coming from Europe (62%), North America (27%) and Asia (6%)).

As an engine itself is completely stateless and independent, adding more computing power is straightforward. Moreover, in order not to deteriorate the reactivity and the performance for handling the clients requests, the code performing the optimization process can easily be realized as a standalone service and can run on distinct servers.

B. Caching Layer

In order to improve the reactivity of the read operations, *Scalia* maintains a distributed (per datacenter) cache layer. Upon a data read, if the data is present in the cache, there is no need to fetch the chunks from the remote providers and reassemble the data object before serving it to the client. Otherwise, the data is reassembled from the chunks, served to the client and stored in the cache. Not only this layer reduces the requests latency, but it also reduces the interactions with the storage providers, resulting in lower costs for the user. In a multi-datacenter setup, the cache has to be invalidated in all datacenters in order to guarantee the consistency of the read operations. The caching layer can be combined and extended by a CDN to reach even better read performance.

C. Database Layer

The database layer of *Scalia* stores the metadata (i.e., the rules set by the end users regarding the durability, availability or vendor lock-in avoidance constraints of their data objects, the public provider settings, the settings of the users' private storage resources) as well as the access history of the data objects (i.e., the statistics). The database can be concurrently accessed by several engines updating the same entry, in all datacenters. As clients' requests are routed to all datacenters indifferently, the database has to be replicated; the classic master-slave replication scheme of traditional databases is not suitable for our multi-master setup, as *Scalia* has to keep working even when a datacenter is down. Moreover, not only the read but also the write operations have to be scalable. Therefore, we consider here a NoSQL database (i.e. Cassandra, http://cassandra.apache.org), which have a better support for multi-datacenter deployment and network/server failures.

1) Concurrency and Conflicts: In a distributed system, a race condition can result in catastrophic situations where concurrent updates for the same entry can lead to data corruption or data loss. To deal with concurrency, two approaches are imaginable in our architecture. The first solution is to use a distributed locking mechanism, such as Zookeeper (http://zookeeper.apache.org), to ensure that an entry is updated only by a single engine at a time. However, because of our multi-datacenter setup, Zookeeper needs to be synchronized among the datacenters and results in higher write operation latency. Even worse, in case of a network partition between the datacenters, Zookeeper is not able to form a quorum and assign locks. To solve this issue, a third party, monitoring all datacenters and assigning the role of a master to one datacenter is required in case of failure. The detailed setup of this architecture is outside the scope of the paper, and will not be discussed here.

Multi version concurrency control (MVCC) [7] is an alternative approach without locks, where an update operation does not delete the old data overwriting it with the new one. Instead, the old data is marked as obsolete and the new version is added, resulting in storing multiple versions of the data with only one being the latest. If an entry is updated concurrently in multiple datacenters, the database will detect the conflict (e.g., employing anti-entropy mechanisms such as vector clocks). The user will be prompted to decide which version is the good one and *Scalia* will remove the other version. Alternatively, *Scalia* can decide by itself to keep only the latest version without asking the end user, however it requires that each engine is time-synchronized (e.g. via NTP).

2) Statistics: The read and write accesses of an object are collected using a distributed and reliable service [e.g., Flume (https:// github.com/cloudera/flume) or Scribe (https://github.com/facebook/scribe)] for efficiently collecting, aggregating, and moving large amounts of log data: a *log agent* residing at each engine continuously reads the logs containing the statistics of the requests handled by the engine, and sends them to one of the *log aggregators*. The latter collect and aggregate the logs before writing them to the database.

The placement algorithm also needs statistics about the objects managed by *Scalia* to take pertinent decisions when there is no access history of new objects, or when it has

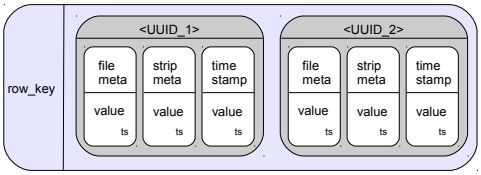

Fig. 10. Concurrent writes: the row_key entry has been updated concurrently, resulting in 2 versions of its metadata. When the conflict is detected, the chunks corresponding to the oldest version are removed from the storage providers, and the oldest version is removed from the database.

to predict the deletion of an object in order to optimize its placement. Those statistics are obtained using map-reduce jobs on the database, so as to aggregate the statistics of each individual objects.

D. Life cycle of read and write operations

Scalia relies on multi version concurrency control (MVCC) to deal with concurrent updates and requires every engine to be time-synchronized (e.g., using NTP) in order to resolve conflicts.

1) Write Operation: During a write operation of an object obj, a user will provide at least the following input through the *Scalia* interface: a container name $obj[container]$, a key $obj[key]$ and the data $obj[data]$. After having decided the optimal set of providers $P(obj)$, *Scalia* splits the data object into $|P(obj)|$ chunks, and stores the latter at the selected storage providers using as key:

$$skey = MD5(obj[container] \mid obj[key] \mid UUID)$$

$UUID$ is a globally unique identifier which prevents concurrent updates to cause data corruption. The metadata of obj is written to the database with $UUID$ as the primary key, as depicted on Figure 10. As row key for writing the metadata, *Scalia* uses:

$$row\_key = MD5(obj[container] \mid obj[key])$$

Table 11 shows an example of metadata stored for an object.

If the write operation is an update, older metadata corresponding to obj is discarded and the corresponding chunks deleted from the providers. The operations are logged and will be processed by the distributed log system, in order to be written in the statistics database. When a conflict is detected by the database in case of concurrent writes, the timestamps are compared, and only the freshest version is kept; the deprecated version of the object is removed from the storage providers and from the statistics database. Note that writing the statistics never conduct to conflicts in the database thanks to an adapted data model, where statistics are always written using globally unique keys.

2) Read Operation: To read an object obj, the end user sends a request to the *Scalia* API with the container name and the object key as parameters. The randomly chosen engine that has received the request checks first if the data is in the cache. If so, then the data is directly returned from the cache. Otherwise, the engine reads the metadata of obj from the database, retrieves the m out of $|P(obj)|$ chunks

Striping metadata	*File metadata*
chunk1: provider_2	name: myvacation.gif
chunk2: provider_5	mime: image/gif
chunk3: provider_7	checksum: ce944a11a4
chunk4: provider_1	size: 342 KB
m: 3	policy: rule 3
skey: a3e229084	container: pictures

Fig. 11. Metadata of the file `myvacation.gif`

from the cheapest (other criteria can be considered) providers, reassembles the data and sends it to the client. The data is also stored in the cache. The operations are logged and stored in the statistics database.

3) Error Handling:

At the providers' side: It may happen that one of the storage providers is not available. If it happens during a write operations, *Scalia* will choose the best placement that does not include the faulty provider. In case of a read operation, if $|P(obj)| > m$, then the data can still be retrieved from the m storage providers available. Recall that m corresponds to the minimum amount of chunks needed to reconstruct a data item. Finally, for a delete operation, the deletion of the chunk residing at a faulty provider is postponed until the provider recovers. As we employ the MVCC approach, incomplete operations do not introduce inconsistencies.

At Scalia side: Within a single datacenter, no layer has a single point of failure. In a multi-datacenter setup, where requests are routed indifferently to each datacenter, the database layer might cause a problem. In fact, thanks to an advanced support of multiple datacenters, the NoSQL database automatically stores a replica in multiple datacenters. Therefore, read requests sent to the *Scalia* API can always be served. Regarding write requests, as long as a single database node is up and running, no operation will fail, and when the second datacenter recovers, the replicas in the various datacenters will be eventually consistent.

E. Private Storage Resources

An interesting property of *Scalia* is the ability to use private storage resources together with commercially available public cloud storage solutions. Corporate storage resources (workstations, servers, NAS, SAN, ...) or dedicated servers can be registered to *Scalia* with a description of their properties: amount and price of available storage, price of incoming and outgoing bandwidth and price per operation. The placement algorithm will take into account these new resources to minimize the costs of storing and serving the user's data. Thanks to *Scalia* and its unified interface, it is straightforward to use local resources up to their capacities, and then use the best suited provider(s) when demand grows.

In order for a private storage resource to be accessible from *Scalia*, a standalone web service needs to be deployed locally on the resource. The web service is a lightweight and standalone web server that offers an authenticated Amazon S3 compatible REST interface to store and retrieve files. The data is stored on the local filesystem or on any distributed/parallel

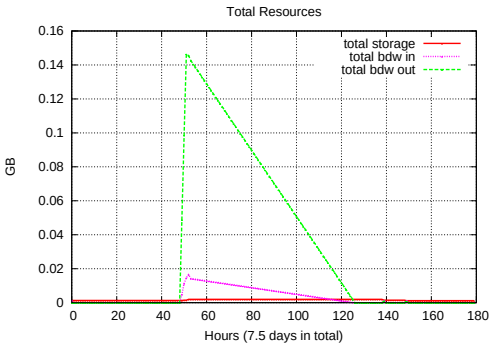

Fig. 12. Slashdot scenario: total amount of resources from the storage providers used by *Scalia* to store and serve the object.

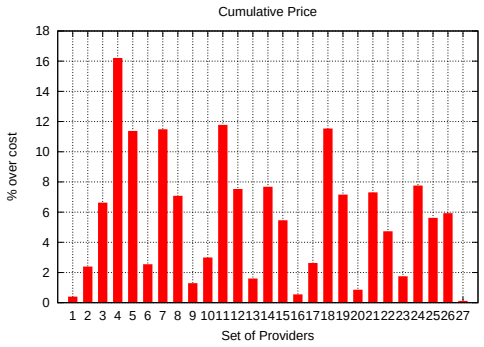

Fig. 14. Slashdot scenario: total cost after a week of the provider sets that satisfy the constraints of the object. *Scalia* is number 27 (cf. Table 13 for labels).

filesystem [e.g., NFS [1], MogileFS (http:// danga.com/mogilefs)] accessible directly from the web service and will never grow beyond the limit set in the properties of the resource. A private token generated by the private resource owner is also registered to *Scalia*, so that only legitimate requests are considered by the web service. The authentication is done by signing the request (i.e., HMAC of the requests parameters using the private token) and to prevent replay attacks, a timestamp is also included in the request. If the data stored is sensitive, the web service can be configured to use SSL/TLS.

IV. Evaluation

A. Experimental Setup

As we mainly discuss the costs involved in several setups, we only present here results coming from a simulator. The availability and durability guarantees of the five public storage providers considered in this evaluation, as well as their pricing policies regarding the costs of resources such as storage, incoming and outgoing bandwidth are described in Table 3. In the following experiments, we consider without loss of generality that Amazon S3 with a specific high durability (i.e., S3(h)) is completely independent from Amazon S3 with a specific low durability (i.e., S3(l)), and there are no correlated failures or any relation between them.

#	*Set of Providers*	#	*Set of Providers*	#	*Set of Providers*
1	S3(h)-S3(l)	10	S3(h)-Azu-Ggl	19	S3(l)-Azu-RS
2	S3(h)-S3(l)-Azu	11	S3(h)-Azu-Ggl-RS	20	S3(l)-Ggl
3	S3(h)-S3(l)-Azu-Ggl	12	S3(h)-Azu-RS	21	S3(l)-Ggl-RS
4	S3(h)-S3(l)-Azu-Ggl-RS	13	S3(h)-Ggl	22	S3(l)-RS
5	S3(h)-S3(l)-Azu-RS	14	S3(h)-Ggl-RS	23	Azu-Ggl
6	S3(h)-S3(l)-Ggl	15	S3(h)-RS	24	Azu-Ggl-RS
7	S3(h)-S3(l)-Ggl-RS	16	S3(l)-Azu	25	Azu-RS
8	S3(h)-S3(l)-RS	17	S3(l)-Azu-Ggl	26	Ggl-RS
9	S3(h)-Azu	18	S3(l)-Azu-Ggl-RS	27	*Scalia*

Fig. 13. Sets of providers.

We compare the cost of multiple static sets of providers with the cost of the dynamic set of providers chosen by *Scalia*. As a baseline, for every sampling period, we compute the *ideal placement*, which corresponds to the cheapest set of provider storage solutions with respect to consumed resources (storage, number of operations, incoming bandwidth, outgoing bandwidth) for handling the load during that period, which is taken as *known* a priori. Given the ideal set of providers for a sampling period, we then compute the corresponding *optimal cost* and the percentage of overhead cost (referred to as "over cost") of the different providers' sets.

B. Slashdot Effect Scenario

In this experiment, we simulate the behavior of the "Slashdot effect", where suddenly an object becomes highly popular and starts to receive a lot of requests. After 2 days (48 hours), the number of read requests goes from 0 to 150 in only 3 hours, and then slowly decreases at the rate of 2 requests per hour. The object stored has size 1MB, a minimum availability of 99.99% and durability of 99.999%. The durability constraint is easily met by only 1 provider; however, the availability constraint requires at least 2 providers. As depicted in Figure 14, *Scalia* is only 0.12% more expensive than the ideal placement. This difference is explained by the cost of the migration of several chunks. *Scalia* uses [S3(h), S3(l), Azure, RS; m:3] before the Slashdot effect. During the requests peak, the cheapest provider set is [S3(h), S3(l); m:1]. When the flash crowd effect is over, *Scalia* chooses [S3(h), S3(l), Azure, Google, RS; m:4] as its provider set. Thanks to the adaptivity of *Scalia*, the best provider set for a given access pattern is always chosen. The best static provider set is a mix of [S3(h), S3(l); m:1] which is 0.4% more expensive, while the worst static provider set [S3(h), S3(l), Azure, Google, RS; m:4] is 16% more expensive than the ideal placement.

C. Gallery Scenario

In this scenario, 200 pictures (250 KB each) have to be stored. The pictures are accessed following the daily pattern of a real website which has around 2500 visitors per day mainly coming from Europe (62%), North America (27%) and Asia (6%). Moreover, the popularity of the pictures follows a Pareto (1,50) distribution. The minimum availability per picture is set to 99.99%.

Ideally, all pictures should not be stored to the same set of providers, because some pictures are popular and the cost of storage is negligible as compared to the cost of outgoing bandwidth. On the other hand, unpopular pictures should be stored to the provider set with the lowest storage cost, while still ensuring the availability and durability constraints.

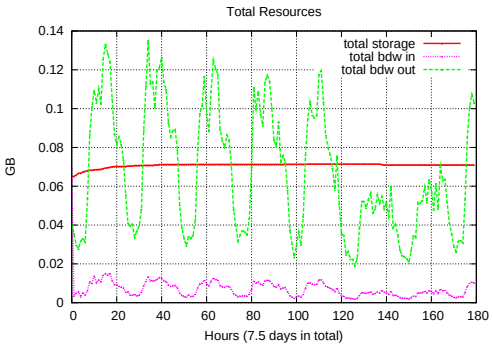

Fig. 15. Gallery scenario: total amount of resources from the storage providers used by *Scalia* to store and serve the pictures.

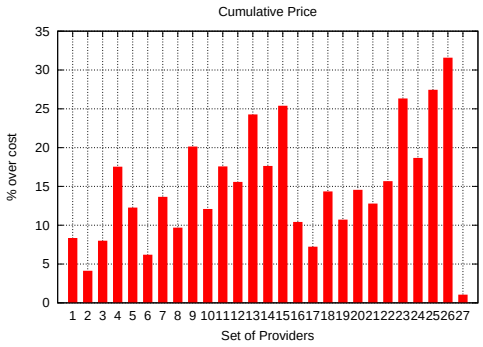

Fig. 16. Gallery scenario: total cost after a week of the provider sets that satisfy the constraints of the object. *Scalia* is number 27 (cf. Table 13).

Figure 15 depicts the total amount of resources used by *Scalia* for storing and serving the pictures from the different storage providers. In Figure 16, *Scalia* is only 1.06% more expensive than the ideal placement and outperforms all the other static sets of providers. The best static set of providers is 4.14% more expensive, while the worst set is 31.58% more expensive than the ideal placement.

The popular pictures mainly use [S3(h), S3(l); m:1], moderately popular pictures use [S3(h), S3(l), Azure; m:2] and unpopular pictures use [S3(h), S3(l), Azure, Google; m:3]. Therefore, it clearly appears that storing all pictures to the same set of providers results in over-charging. The adaptivity of *Scalia* dynamically finds the most cost-efficient placement of an object based on its access pattern. Therefore, an end user does not need to decide a fixed placement per data object by guessing the access pattern of the object.

D. Adding Storage Resources

We now consider a scenario where a new object of 40 MB needs to be stored every 5 hours. Unlike preceding scenarios where the availability constraint was important, here the data owner wants to avoid vendor lock-in and therefore each object has to be stored at 2 different providers at least. At hour 400 a new storage provider *CheapStor* is registered in the system and offers an attractive storage alternative: 0.09$ per GB of storage, 0.1$ per GB of bandwidth in, 0.15$ per GB of bandwidth out and 0.01$ for 1K of operations.

Before hour 400, *Scalia* stores the objects using [S3(h), S3(l), Azure, Google, Rackspace; m:4]. After the new provider has been registered, *Scalia* migrates the already stored objects and stores the new objects to [S3(h), S3(l), Azure, CheapStor, Rackspace; m:4]. The total amount of resources used in this experiment is shown in Figure 17.

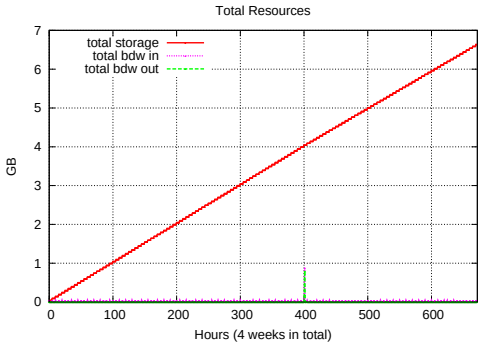

Fig. 17. Adding a public storage provider: total amount of resources used by *Scalia* to store the backup objects to the storage providers.

In this scenario, *Scalia* dynamically adapts to the changing conditions (a new provider has shown up) and is only 0.35% more expensive than the ideal placement. The best static placement [S3(h), S3(l), Azure, Google, Rackspace; m:4] is not able to take into account the new provider, and therefore costs 7.88% more than the ideal placement, while, the worst static placement is 96.35% more expensive!

E. Active repair

In the case of a transient failure of a cloud storage provider, *Scalia* may adopt two strategies to cope with the unavailability of providers: either do nothing and simply wait for the provider to recover, or move the chunks hosted at the faulty provider to another provider. However, the latter procedure comes at a relatively high cost: in order to move the chunk of the faulty provider, the data object needs to be reconstructed from the remaining chunks and split again into chunks. Depending on the available providers, the threshold m of the most cost-effective providers set may be different. In that case, all chunks need to be re-written. If m is the same, then only the faulty chunk needs to be written, which corresponds to the cheapest case.

As in Section IV-D, we consider a scenario where a new object of 40 MB needs to be stored every 5 hours. At hour 60, one of the provider, S3(l), has a transient failure and is not reachable anymore. At hour 120, the provider is again up and running.

Scalia is compared to the static provider set [S3(h), S3(l), Azu; m:2]. Before hour 60 and after hour 120, *Scalia* uses [S3(h), S3(l), Azu; m:2] as well. During the unavailability of S3(l), *Scalia* uses another provider to store the unreachable chunk: [S3(h), Ggl, Azu; m:2]. However, in the static provider set, the unreachable chunk cannot be moved to another provider, and therefore the data needs to be split into only 2 chunks, resulting in using [S3(h), Azu; m:1] during the failure period. Figure 18 shows the cost difference of active repair between the fixed and the dynamic sets of providers.

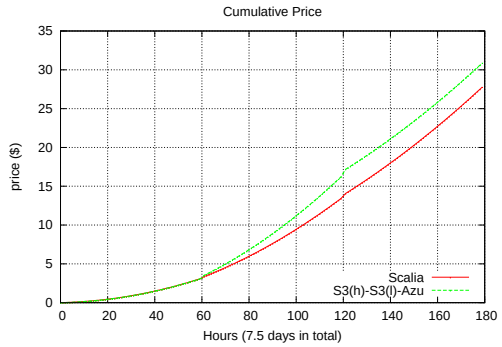

Fig. 18. *Scalia* versus a fixed set of providers during active repair.

V. Related Work

Scalia was inspired by RACS [2], which employs RAID at the cloud storage level, making also use of erasure codes [3] instead of full replication [4]. However, RACS does not adapt data placement to different conditions to meet any optimization objectives, as opposed to our work. HAIL [8] distributes redundant blocks of a file across multiple servers, while allowing a client to make sure that the file is not corrupted even in the case of a server compromise. RAID-like techniques have already been used by several P2P storage systems [9], [10], [11] to ensure durability and availability of data. Storage providers like Wuala (http://wuala.com) use an hybrid model where data is split into redundant blocks at client side (via erasure coding); the blocks are distributed to other end users following a P2P approach and also sent to the servers managed by the provider. This approach increases durability of the data, while decreasing the storage costs of the provider. Other commercial providers like Cleversafe (http://cleversafe.com) also make use of erasure coding to disperse data blocks to several geographical distinct regions over the world, providing very high durability guarantees. A static approach for matching performance requirements to cloud resources from multiple providers was proposed in [14]. All the aforementioned approaches do not improve the placement of the data objects according to their access pattern.

Commercial network appliances or servers [Cloud AFS (http://gladinet.com), Nasuni Filer (http://nasuni.com)] residing at the customer, called cloud storage gateway, can also serve as intermediaries to multiple cloud storage providers. While they include storage features such as caching, backup, recovery, encryption or de-duplication, these systems do not take into account the access pattern of the data nor its expected lifetime to continuously choose the optimal providers set based on any criteria, as opposed to *Scalia*.

Also, several libraries (http://jclouds.org, http:// incubator.apache.org/libcloud) for accessing public cloud storage and cloud computing infrastructures with a unified interface are quickly emerging, thus showing the increased need of avoiding vendor lock-in.

Finally, extensive previous work [12], [13] is available in the area of job scheduling in computational grids, so as to minimize the cost for the end-users, while satisfying the performance constraints. However, most of these works depend on prior knowledge of the detailed computational cost of a new job and the job placement is fixed. In *Scalia*, data placement is adaptive to the various pricing and resource conditions, so as to dynamically find the optimal data placement.

VI. Conclusions

We have presented *Scalia*, a system that continuously optimizes the placement of data stored at multiple cloud providers, based on their access statistics. *Scalia* mediates data placement across multiple public cloud providers and private cloud resources. It helps the data owners to avoid vendor lock-in and satisfy certain availability and durability constraints in a cost-effective way. We described in detail the various layers of our approach and our scalable mechanism for adaptive data placement. By extensive simulation experiments, we proved that our solution finds the optimal (e.g., cheapest based on the access statistics) data placement for dynamically changing data access patterns and when different cloud storage solutions are available. The evaluation of the latency overhead and the scalability of our prototype implementation is left for future work.

References

[1] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon, "Design and Implementation or the Sun Network Filesystem", 1985.

[2] H. Abu-Libdeh, L. Princehouse and H. Weatherspoon, "RACS: A Case for Cloud Storage Diversity", in *Proc. of SOCC*, Indianapolis, USA, 2010.

[3] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright and K. Ramchandran, "Network Coding for Distributed Storage Systems", in IEEE Transactions on Information Theory, Vol. 56, Issue 9, Sept. 2010.

[4] H. Weatherspoon and J. Kubiatowicz, "Erasure Coding Vs. Replication: A Quantitative Comparison", in Revised Papers from IPTPS'01, Springer-Verlag, London, UK, 2002.

[5] R. Rodrigues and B. Liskov, "High Availability in DHTs: Erasure Coding vs. Replication", in 4th International Workshop IPTPS, Ithaca, New York, 2005.

[6] H. Kellerer, U. Pferschy and D. Pisinger, "Knapsack Problems", *Springer Verlag*, 2004.

[7] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", ACM Computing Surveys, 1981.

[8] K. D. Bowers and A. Juels and A. Oprea, "HAIL: a high-availability and integrity layer for cloud storage", in Proceedings of the 16th ACM conference on Computer and communications security, Chicago, Illinois, USA, 2009.

[9] B-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris, "Efficient Replica Maintenance for Distributed Storage Systems", in *Proc. of the NSDI*, May 2006.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris and I. Stoica, "Wide-area cooperative storage with CFS", in *Proc. of the SOSP*, 2001.

[11] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: the OceanStore Prototype", in *Proc. of the FAST*, March 2003.

[12] D. Abramson, J. Giddy, and L. Kotler, "High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?", in *Proc. of the IPDPS*, 2000.

[13] J. Brunelle, P.Hurst, J.Huth, L.Kang, C.Ng, D.C.Parkes, M.Seltzer, J.Shank S.Youssef, "Egg: an extensible and economics-inspired Open grid computing platform", in *Proc. of the Grid Economics Workshop (GECON)*, 2006.

[14] A. Ruiz-Alvarez and M. Humphrey, "An automated approach to cloud storage service selection", In Proc. of ScienceCloud Workshop, 2011.